CSC290/420 Machine Learning Systems for Efficient AI
ML Programs as Loop-Intensive Code

Sreepathi Pai

October 6, 2025

URCS

Outline

Loop-Intensive Code

Single Loop Transformations

Multi-Loop Transformations

Autotuning: A brief introduction

Outline

Performance of Programs

- "Hot spots" in a program are those that consume the most amount of execution time
  - Often, a very small portion of the code (80/20 rule)
- Almost invariably, this hot spot lies in a loop
  - Why?

Loops in ML Programs

- ML programs uses indexed data structures
  - arrays, matrices, tensors
- Loops are used to access these data structures and perform operations on them
- Nearly all operators in an ML program are composed of loops
  - exceptions include data transformation operators

Loop Optimizations

- "Front-end" optimizations
  - Generate enough work to keep the processor pipeline/cores full of work
  - Reduce branches
  - i.e., increase (useful) ILP, MLP, and TLP.
- "Back-end" optimizations
  - Reduce expense of work performed by the compute and memory units
  - Use appropriate functional units (e.g., SIMD units)
  - And memory-related optimizations to increase cache hit rate (or locality)

Loop Transformations

- Scientific code running on supercomputers is also loop-intensive code
- Multiple decades of research on automatically transforming loops to optimize them
- Specialized compilers for Scientific code, usually vendor-provided
  - IBM XLC (gone), Intel Compiler (icc, gone), NVIDIA/Portland Group
  - limited support in GCC (Graphite) and LLVM (Polly)
- Disclaimer: this class isn't about understanding theory of loop transformations
  - that is content for an advanced compilers class (CSC255/455++)

Loop Transformations for ML Programs

- Modern ML compilers inherit from this tradition
  - Google's XLA
  - Apache TVM
  - OpenAI Triton
  - LF IREE
- All of them perform loop transformations
  - Mostly automatically, but also under control of programmer
- Nearly all research on optimizing ML programs invariably boils down to loop optimization
  - I'm kidding, but only just.

Outline

Loop Unrolling

```
for(i = 0; i < N; i++) {
   c[i] = a[i]*b[2*i];
}
```

```
for(i = 0; i < N/4; i+=4) {
   c[i  ] = a[i  ]*b[2*i];
   c[i+1] = a[i+1]*b[2*(i+1)];
   c[i+2] = a[i+2]*b[2*(i+2)];
   c[i+3] = a[i+3]*b[2*(i+3)];
}
```

- Replicate body of loop U times
  - U is unroll factor
- Reduce iteration count of loop U times
- Resources: Registers, Functional Units
- Reduces number of instructions executed
  - Loop overhead

Loop Splitting/Peeling

```
for(i = 0; i < N/4; i+=4) {
   c[i  ] = a[i  ]*b[2*i];
   c[i+1] = a[i+1]*b[2*(i+1)];
   c[i+2] = a[i+2]*b[2*(i+2)];
   c[i+3] = a[i+3]*b[2*(i+3)];
}

for(; i < N; i++) {
   c[i] = a[i] * b[2*i];
}
```

- Break loop into multiple parts
  - usually to handle irregular portions
  - or meet alignment restrictions for vectorization

Loop Vectorization

```
// fictitious ISA
for(i = 0; i < N/4; i+=4) {
   simd_value_4 c;
   c = simd_mul_4(a[i], b[2*i])
   simd_store_4(c[i], c);
}

for(; i < N; i++) {
   c[i] = a[i] * b[2*i];
}
```

- Combine loop iterations to use SIMD instructions automatically
  - Only possible if loop has no dependence between iterations
- Legality of transformation can be checked by some compilers
  - Others simply follow the user's command

Loop Parallelization

```
for(i = thread_start;
    i < (condition) && i < N;
    i += thread_increment) {
   c[i] = a[i]*b[2*i];
}
```

- Divide loop iterations across multiple compute units
  - usually abstracted as threads
  - so thread-level parallelism, TLP
- Can be combined with SIMD and ILP

Distributing Loop Iterations

- Block Distribution
  - `thread_items = (N+num_threads - 1)/num_threads`
  - `thread_start = thread_id * thread_items`
  - `thread_increment = 1`
  - `condition` is `thread_start + thread_items`
- Round-Robin
  - `thread_items = (N+num_threads - 1)/num_threads`
  - `thread_start = thread_id * thread_items`
  - `thread_increment = thread_items`
  - `condition` is empty

Software Pipelining

```
for(i = 0; i < N; i++) {
   a = 2 * B[i];
   b = a + 1;
   c = b / 2;
}
```

- Execute multiple iterations of a loop in parallel
- Beneficial when loop body contains a long dependence chain
  - but iterations are independent
- Best understood as unrolling and reordering of instructions within loop

Software pipelining example

```
for(i = 0; i < N/3; i+=3) {
   a = 2 * B[i];
   b = a + 1;
   c = b / 2;

   a1 = 2 * B[i+1];
   b1 = a1 + 1;
   c1 = b1 / 2;

   a2 = 2 * B[i+2];
   b2 = a2 + 1;
   c2 = b2 / 2;
}
```

Software pipelining example (contd.)

```
for(i = 0; i < N/3; i+=3) {
   a = 2 * B[i];
   a1 = 2 * B[i+1];
   a2 = 2 * B[i+2];

   b = a + 1;
   b1 = a1 + 1;
   b2 = a2 + 1;

   c = b / 2;
   c1 = b1 / 2;
   c2 = b2 / 2;
}
```

- Is software pipelining beneficial for:
  - out-of-order processors?
  - in-order processors?

Other Transformations

- Loop skewing
  - Expose parallelism among iterations
- And many others

Outline

Multiple Loops

- Here, multiple loops can be independent loops or nested loops
  - Independent: `for(...) {} for(...) {}`
  - Nested: `for(...) { for(....) {} }`
- For loop transformations, nested loops often need to be "properly nested"
  - Only the innermost loop contains code

Loop Interchange

```
for(i = 0; i < M; i++)
  for(j = 0; j < N; j++)
    for(k = 0; k < K; k++)
      c[...] = ...
```

```
for(i = 0; i < M; i++)
  for(k = 0; k < K; k++)
    for(j = 0; j < N; j++)
      c[...] = ...
```

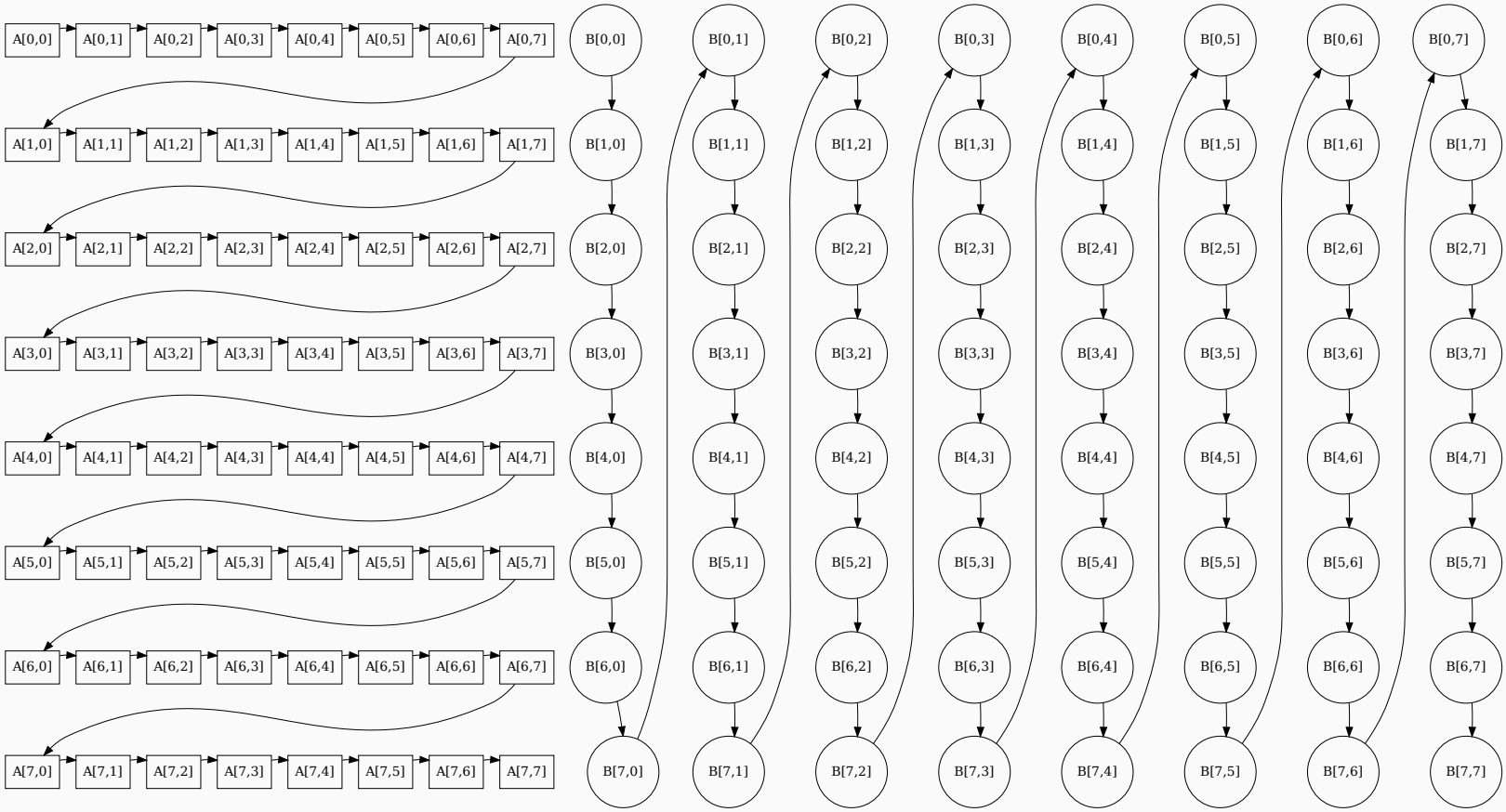
- Change order of nested loops
- Primarily a locality-enhancing transformation
  - but may expose additional parallelism

Loop Blocking/Tiling

```
// matrix transpose
for(i = 0; i < M; i++)
   for(j = 0; j < N; j++)
      b[j][i] = a[i][j]
```

- Convert a loop (or a set of loops) to access data in a "blocked" manner
- Primarily a locality-enhancing transformation
- In the loop above, reflect on the locality of `b[j][i]`

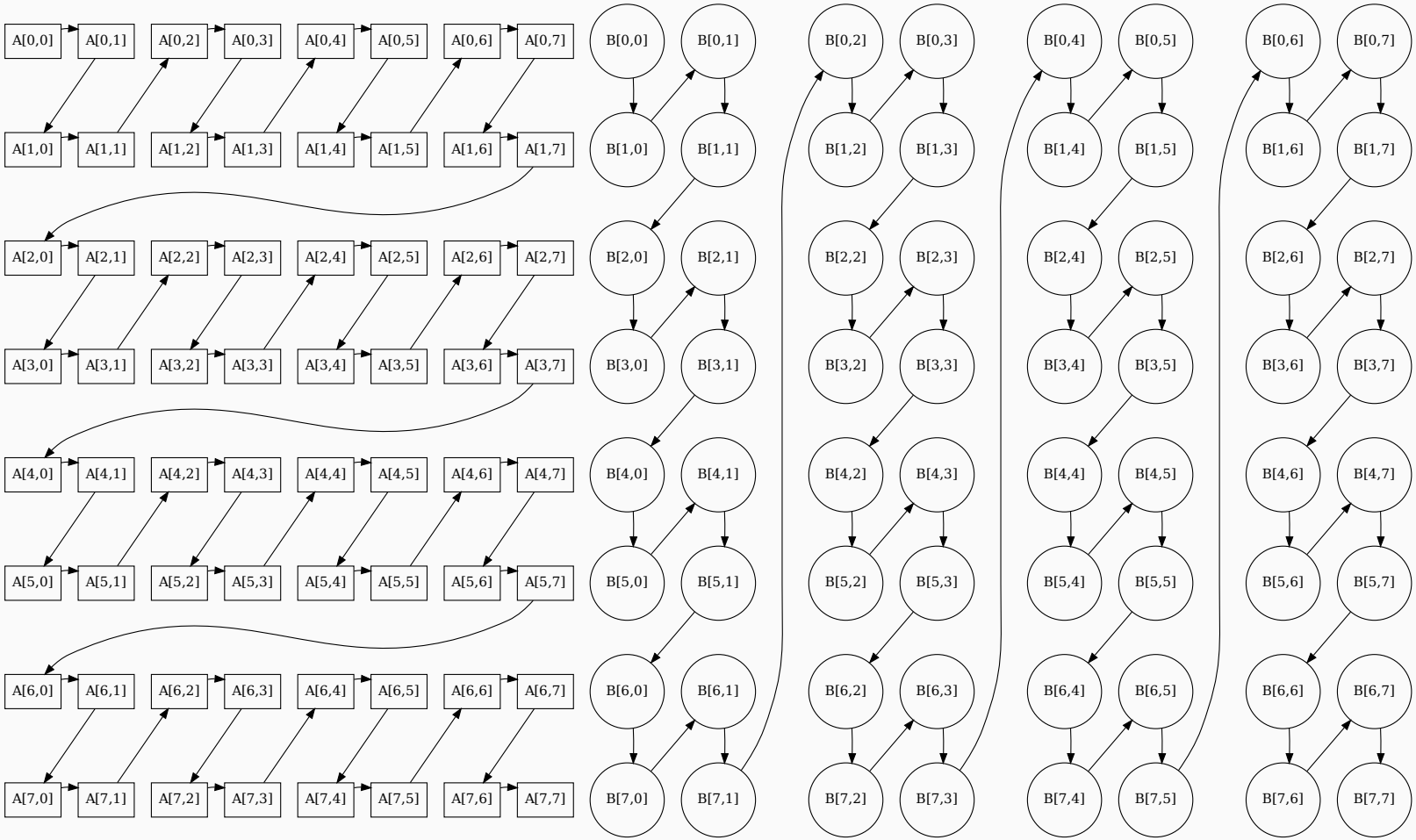
Locality



- Assume row-major ordering
- Each cache line contains two elements
- Cache capacity is four cache lines
- Arrows show order of access within each matrix

Loop after tiling

```
for(i = 0; i < M; i++)
   for(j = 0; j < N; j+=B)
      for(ii = i; ii < N && ii < (i + B); ii++)
         for(jj = j; jj < N && jj < (j + B); jj++)
            b[jj][ii] = a[ii][jj]
```



Loop Fusion

```
for(i = 0; i < N; i++)
   A[i] = B[i]

for(i = 0; i < N; i++)
   A[i] = 2*A[i]
```

```
for(i = 0; i < N; i++)
   A[i] = B[i]
   A[i] = 2*A[i]
```

- Combine two loops into a single loop
- Reduces memory writes/communication
- Enhances cache locality

Other transformations

- Loop unroll-and-jam (fuse)
  - for vectorization
- Software pipelining across producer/consumer loops
  - execute loops on different cores

Frameworks

- Unimodular framework
  - Classic framework supported by many HPC compilers
- Polyhedral framework
  - Modern framework based on affine representations and transformations

Outline

Parameters

- Loop transformations require parameters
  - Which loop to unroll?
  - Unrolling depth
  - Tile sizes
  - Which loops to fuse?
- No exact theory exists to compute these parameters
  - to the best of my knowledge
  - Program + Machine + Input determine values that deliver optimal performance

Empirical Search

- Current method of choice is to empirically search the space of parameters
- Brute-force search
  - Intractable
- Sparse search
  - Search over limited set of parameters (most current ML frameworks)
  - Search over full space, but "intelligently" (mostly HPC software)

Autotuning

- Classic technique in high-performance computing to achieve best performance
- Now used, in a limited way, by ML frameworks
- Heuristics-based search
- Empirical evaluation

Autotuning Algorithm (Brute Force/Limited Set)

```
for P in parameter_space:
    compile program with P
    run program and measure runtime
    is runtime less than best known?
        yes: remember P as best_P

return best_P
```

Autotuning Algorithm (Heuristic)

```
while not out of time:
    pick P from parameter space using heuristic
    compile program with P
    run program and measure runtime
    is runtime less than best known?
        yes: remember P as best_P

return best_P
```

- The heuristic uses past experience to find new P values
- Examples of heuristics: evolutionary search, simulated annealing, etc.

Loop-based Workflows

- Loop-based code + Optimizers + Autotuning
  - Handwritten code, low-level
  - Tensor code, high-level
  - Optimizers: IREE, PLUTO
- Domain-specific Language + Schedule + Autotuning
  - Halide
  - TVM
- Tiled DSL + Autotuning
  - OpenAI Triton
  - Google Pallas
  - NVIDIA cuTile / NVIDIA Tilus